



Tecnologia em Análise e Desenvolvimento de Sistemas
Tópicos Especiais em Tecnologia da Informação
Prof. Rafael Henrique Dalegrave Zottesso

Programação para Web com Python e Django

Links úteis

- Python 3: <https://www.python.org/download/releases/3.0/>
- Django: <https://www.djangoproject.com/>

Django

- Por onde começar: <https://www.djangoproject.com/start/>
- Documentação: <https://docs.djangoproject.com/pt-br/2.2/>
- Tutorial: <https://docs.djangoproject.com/pt-br/2.2/intro/tutorial01/>

Passos para rodar o projeto na aula

1. Baixar projeto: *git clone http://....*
2. Ativar o ambiente virtual: *source /opt/virtualenv/django2019/bin/activate*
3. Navegar no terminal até a pasta do seu projeto que tem o arquivo *manage.py*
4. Iniciar o servidor: *python manage.py runserver*

Documentação das Views baseadas em classes

<https://docs.djangoproject.com/pt-br/3.1/ref/class-based-views/generic-editing/>

Criando um usuário administrador

Primeiro temos que criar um usuário que possa acessar o site de administração. Rode o seguinte comando:

```
$ python manage.py createsuperuser
```

Acesse: <http://localhost:8000/admin/>

Criando modelos

No arquivo `models.py` de alguma de suas aplicações/módulos, crie uma classe simples com os seguintes atributos:

```
from django.db import models

# Create your models here.
class Estado(models.Model):
    sigla = models.CharField(max_length=2)
    nome = models.CharField(max_length=50)

    def __str__(self):
        return self.sigla + " - " + self.nome

class Cidade(models.Model):
    nome = models.CharField(max_length=50)
    estado = models.ForeignKey(Estado, on_delete=models.PROTECT)

    def __str__(self):
        return self.nome + " - " + self.estado.sigla

class Pessoa(models.Model):

    nome = models.CharField(max_length=50, verbose_name="Qual seu nome?",
help_text="Digite seu nome completo")
    nascimento = models.DateField(verbose_name='data de nascimento')
    email = models.CharField(max_length=100)

    cidade = models.ForeignKey(Cidade, on_delete=models.PROTECT)

    def __str__(self):
        return self.nome + ' - ' + str(self.nascimento)
```

A classe Cidade é uma extensão da classe “Model” que está dentro do pacote “models”. Dentro dela você pode criar vários atributos.

Existem diversos tipos (<https://docs.djangoproject.com/pt-br/2.2/ref/models/fields/#field-types>), alguns mais comuns são:

- models.**CharField**(...) - Campo de texto comum (input do tipo texto)
- models.**TextField**(...) - Campo de texto grande (textarea)
- models.**DateField**(...) - vai ser um campo de data
- models.**DateTimeField**(...) - vai ser um campo de data e hora
- models.**IntegerField**(...) - valores inteiros
- models.**DecimalField**(...) - valores decimais

Propriedades (<https://docs.djangoproject.com/pt-br/2.2/ref/models/fields/#field-options>):

- **max_length** - tamanho máximo de caracteres.
- **verbose_name** - Nome (label) que vai aparecer no formulário para identificar o campo/atributo.
- **help_text** - Texto extra de ajuda para ser mostrado com o “widget” do formulário. É útil para documentar mesmo que seu campo não seja usado em um formulário.
- **null** - Se True, o Django pode usar valores nulos (null) no banco de dados. Padrão é False
- **blank** - Se True, é permitido o campo estar em “branco” (vazio). O padrão é False.
- **default** - O valor padrão para o campo.
- **unique** - Se True, aquele atributo vai ser único no banco de dados. O padrão é False.

Para os campos de **data**, ainda deve-se preencher dois atributos:

- **auto_now=True** - Adiciona a data/hora toda vez que o objeto é salvo no banco. Geralmente, usado para verificar a última data/hora de modificação do objeto.
- **auto_now_add=True** - Adiciona a data/hora toda vez que o objeto é criado no banco. Geralmente, usado para guardar a data que o objeto foi criado.

Campos relacionais (<https://docs.djangoproject.com/pt-br/2.2/ref/models/fields/#foreignkey>):

- models.**ForeignKey**(Classe, on_delete=models.PROTECT) - chave estrangeira representada por um select no formulário. Precisa informar a classe do relacionamento e o modo de relação quando algum objeto da Classe for excluído (PROTECT, CASCADE, etc)

Opções adicionais de configuração de modelos (classe **Meta**):

<https://docs.djangoproject.com/pt-br/2.2/ref/models/options/>

Sincronizando seu banco de dados com o models

Ao executar makemigrations, você está dizendo Django que você fez algumas mudanças em seus modelos (neste caso, você fez novas modificações) e que você gostaria que as alterações sejam armazenadas como uma migração. Para isso, execute:

```
$ python manage.py makemigrations seu_modulo_app
```

Agora rode o *migrate* para criar essas tabelas dos modelos no seu banco de dados:

```
$ python manage.py migrate
```

Acesse: <http://localhost:8000/admin/>

Ative os modelos criados no painel do administrador

No arquivo admin.py, importe todos os modelos do seu módulo e os ative no site administrativo:

```
from .models import *

admin.site.register(Estado)
admin.site.register(Cidade)
admin.site.register(Pessoa)
```

Acesse: <http://localhost:8000/admin/>

Criando uma método para cadastrar alguma coisa

No arquivo `views.py` é necessário importar todos os modelos, assim como fizemos no admin:

```
from .models import Estado, Cidade, Pessoa
```

Também já iremos importar um método para redirecionar o usuário depois de efetuar o cadastro:

```
from django.urls import reverse_lazy
```

Por fim, vamos importar as Views (classes) que utilizaremos como “Pai” para nossas telas de cadastro (inserir, alterar e excluir):

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
```

Agora podemos criar nossas classes conforme o exemplo abaixo:

```
class EstadoCreate(CreateView):  
    model = Estado  
    fields = ['sigla', 'nome']  
    template_name = 'formulario.html'  
    success_url = reverse_lazy('index')
```

```
class CidadeCreate(CreateView):  
    model = Cidade  
    fields = ['nome', 'estado']  
    template_name = 'formulario.html'  
    success_url = reverse_lazy('index')
```

Agora basta fazer esse mesmo esquema para todos as classes que você deseja criar um formulário de cadastro no seu projeto.

Criando um formulário padrão para usar em diversos modelos

Você pode criar um arquivo na pasta “templates” chamado “formulario.html”, por exemplo. Dentro dele ajuste o layout como preferir e crie um formulário conforme abaixo:

```
<form method="post">  
    {% csrf_token %}  
  
    {{ form.as_p }}  
  
    <button type="submit" class="btn btn-success">  
        Cadastrar  
    </button>  
</form>
```

Não é necessário fazer validação de campos obrigatórios.

Criando a URL para cadastrar um modelo

No arquivo `urls.py`, crie um novo registro apontando para o método que você criou lá no `views.py`:

```
path('cadastrar/estado/', EstadoCreate.as_view(), name="cadastrar-estado"),  
path('cadastrar/cidade/', CidadeCreate.as_view(), name="cadastrar-cidade"),
```

Utilizando o formulários do Bootstrap4

Existe um plugin (<https://django-crispy-forms.readthedocs.io/en/latest/>) que faz a geração do formulário e das mensagens de erro de validação conforme o Bootstrap 4. Para ativá-lo, vá no `settings.py`, procure por `INSTALLED_APPS` e adicione o módulo do plugin. Adicione, também, a constante no `settings` para definir o tipo padrão de templates logo depois do `INSTALLED_APPS`:

```
INSTALLED_APPS = [  
    ...  
    'crispy_forms',  
]
```

```
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

No seu “`formulario.html`”, por exemplo, carregue o plugin depois de dar um load no static:

```
{% load crispy_forms_tags %}
```

No código HTML do formulário altere `{{ form.as_p }}` para `{{ form|crispy }}`

Agora atualize a página.

Criando uma tela para atualizar

O processo para atualizar registros é muito parecido com o CreateView. Portanto, altere apenas a classe que o método lá no views.py estende para **UpdateView**. O restante é a mesma coisa. Por exemplo:

```
class EstadoUpdate(UpdateView):  
    model = Estado  
    fields = ['sigla', 'nome']  
    template_name = 'formulario.html'  
    success_url = reverse_lazy('index')
```

Para criar a URL é necessário passar o ID do registro que será alterado e qual o tipo dele. Por padrão, todas as tabelas no Django criam um campo "id" do tipo inteiro. Então, crie uma nova url no seu urls.py:

```
path('atualizar/estado/<int:pk>/', EstadoUpdate.as_view(),  
     name="atualizar-estado"),
```

Acesse <http://localhost:8000/atualizar/estado/1/>

Criando tela para excluir

Mesmo esquema, porém estendendo a classe `DeleteView` e não é necessário colocar o atributo `"fields"`:

```
class EstadoDelete(DeleteView):  
    model = Estado  
    template_name = 'formulario.html'  
    success_url = reverse_lazy('index')
```

Exemplo de url:

```
path('excluir/estado/<int:pk>/', EstadoDelete.as_view(),  
     name="deletar-cidade"),
```

Usando o mesmo form, vai parecer que você está inserindo/salvando um registro. Então, crie um novo formulário personalizado com uma mensagem de exclusão:

```
<form method="post">  
    {% csrf_token %}  
    <p>Tem certeza que deseja excluir o registro "{{ object }}"?</p>  
    <input type="submit" class="btn btn-danger" value="Sim, excluir.">  
</form>
```

Listando objetos do banco de dados

<https://docs.djangoproject.com/pt-br/2.2/ref/class-based-views/generic-display/>

Importe o método ListView:

```
from django.views.generic.list import ListView
```

Crie um método no views.py que estende ListView e informe qual o modelo será usado e o template:

```
class EstadoList(ListView):  
    model = Estado  
    template_name = 'adocao/listar_estados.html'
```

Agora a tela de listagem você tem que fazer uma para cada por causa dos nomes dos atributos de cada modelo. Exemplo:

```
<table class="table table-striped">  
  
    <tr>  
        <th>ID</th>  
        <th>Sigla</th>  
        <th>Nome</th>  
        <th>Opções</th>  
    </tr>  
  
    {% for estado in object_list %}  
  
    <tr>  
        <td>{{estado.pk}}</td>  
        <td>{{estado.sigla}}</td>  
        <td>{{estado.nome}}</td>  
        <td>  
            editar excluir  
        </td>  
    </tr>  
  
    {% empty %}  
    <tr>  
        <td colspan="3">Nenhuma cidade cadastrada!</td>  
    </tr>  
    {% endfor %}  
  
</table>
```

Por fim, a url para acessar a página criada:

```
path('listar/estados/', EstadoList.as_view(), name="listar-estados"),
```

Agora, para colocar os links para as páginas de “alterar” e “excluir”, modifique a tabela adicionando a URL no formato do Django, passando o “pk” como parâmetro:

```
<a href="{% url 'editar-estado' estado.pk %}" class="btn btn-warning  
btn-sm">Editar</a>  
<a href="{% url 'excluir-estado' estado.pk %}" class="btn btn-danger  
btn-sm">Excluir</a>
```

Um parâmetro pode ser passado para uma URL da seguinte forma:

```
{% url 'editar-estado' objeto.atributo %}
```

Um link para adicionar um estado também pode ser adicionado no **listar_estado.html**:

```
<a href="{% url 'cadastrar-estado' %}">Adicionar estado</a>
```

Por fim, altere o **EstadoCreate**, **EstadoUpdate** e **EstadoDelete** para redirecionar o usuário para a lista e não mais para o index:

```
success_url = reverse_lazy("listar-estados")
```